Char-RNN with Multiple Optimizers

Yutao Ye **y8ye@ucsd.edu**
University of California, San Diego

Abstract

Char-RNN is a particular type of RNN specialized in character and language modeling. Its purpose is to predict the next character when given the current character and the previous hidden state. After the training, after giving the initial character and initial hidden state, I will be able to generate the whole sequence iteratively. By using different hyper-parameters along with the Anna Karenina text [1], and a Linux text from "Character-level Recurrent Neural Networks in Practice: Comparing Training and Sampling Schemes". Neural Computing and Applications (2018).[2] I am looking forward to a high-accuracy Char-RNN model. The hyper-parameters I use includes different training rate, and I change the training dataset, and trained weight for better model performance. I also compare the performance of using different hidden states, like Tanh, leaky ReLU, and SELU. I also test the performance on different types of datasets. Finally, I found a meaningful outcome based on the training above.

Keywords: Char-RNN, SELU, RNN, Latex

1. Introduction

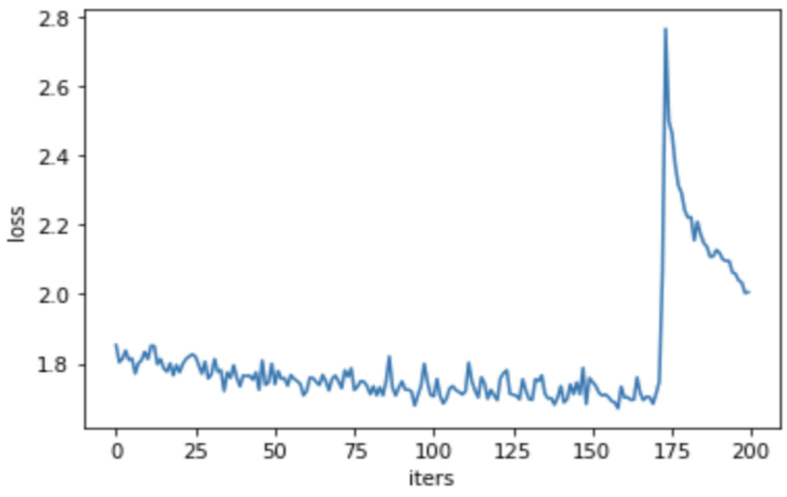
The field of natural language processing has been developed at a skyrocketing speed, and NLP is used everywhere in our lives. When we use the keyboard of our cell phone, the keyboard often successfully predicts what character we will type next. That's because the phone is learning your frequent word combination. Other examples include translation software, search engines, etc. Text prediction or text generation is one of the most interesting topics that attract me. Before I was introduced to Char-RNN, I always wondered if the process was like repeating the whole word sets. Or only based on my preference? Char-RNN mainly focuses on the inherent dependencies and structures of the characters instead of the individual words, which makes the prediction have better performance. What makes it powerful is the function of many to many, according to “The Unreasonable Effectiveness of Recurrent Neural Networks” by Andrej Karpathy, RNNs combine the input vector with their state vector with a fixed (but learned) function to produce a new state vector. This can in programming terms be interpreted as running a fixed program with certain inputs and some internal variables. (Karpathy 2015) [3] The basic process of RNN is to compute the hidden state and predict the output. After being given the initial hidden state and input, RNN will compute the current hidden state and generate output, and then iterate again and again. In this paper, I will try to explore the capabilities and limitations of Char-RNN with multiple datasets. I am also interested in the performance of Char-RNN within different types of datasets.

2. Method

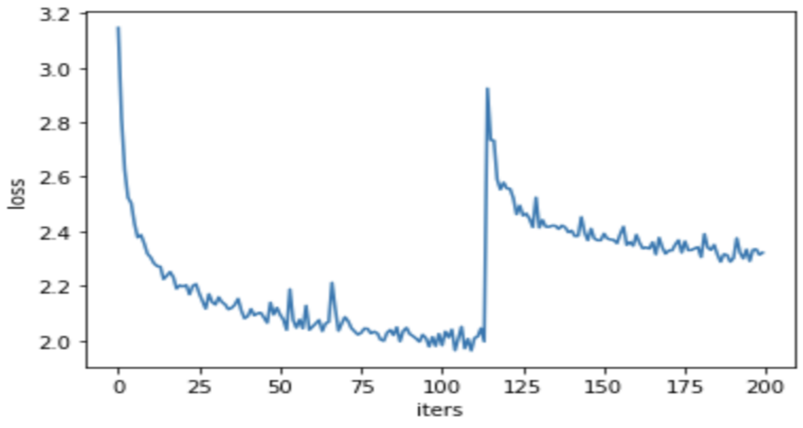
I will use the Jupyter Notebook from Cogs 185 homework 3 [5] with different settings. I will have a character sequence {ct} = {c1, c2..., cT −1, cT } where each ct ∈ A and A is the set of an alphabet. I will set the initial hidden state h0, initial input character c1, and multiple training weights. I will convert ct into one-hot encoding xt, then utilize the previous hidden state to calculate the current hidden state and predict the output. I will use the softmax function to come up with probability to decide the predicted character. Then use the predicted character as the next input character and loop again. (Homework Assignment 3) During training, I will also use the cross-entropy loss to find out the loss and optimize the average loss. There are two datasets I will be using, the Anna Karenina text, and a Linux text. The different settings I will use are different amounts of hidden states from 50 to 200, different training rates, different hidden states, and different input sizes.

3. Experiment

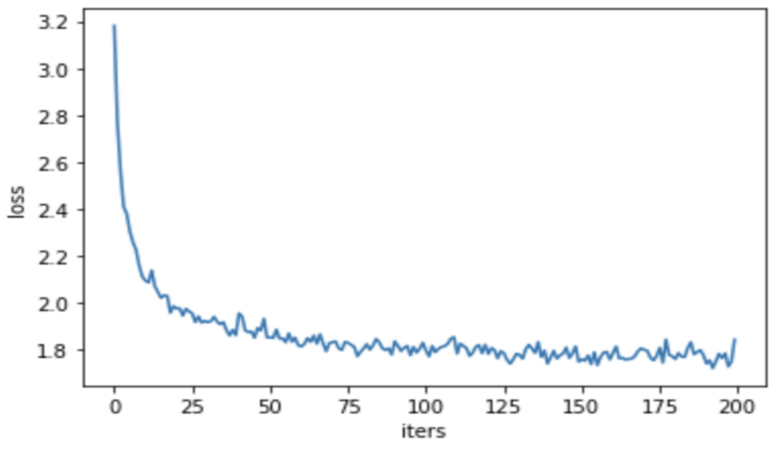
The first dataset I work with is the Anna Karenina text. The number of iterations is 20,000, with 100 hidden states, and 100-dimension input, and output. I am using Tanh as an activation function. After training, there was a sharp increase in loss after 175 iterations, it went from 1.8 to 2.7, then decrease gradually. The whole process was not consistent.



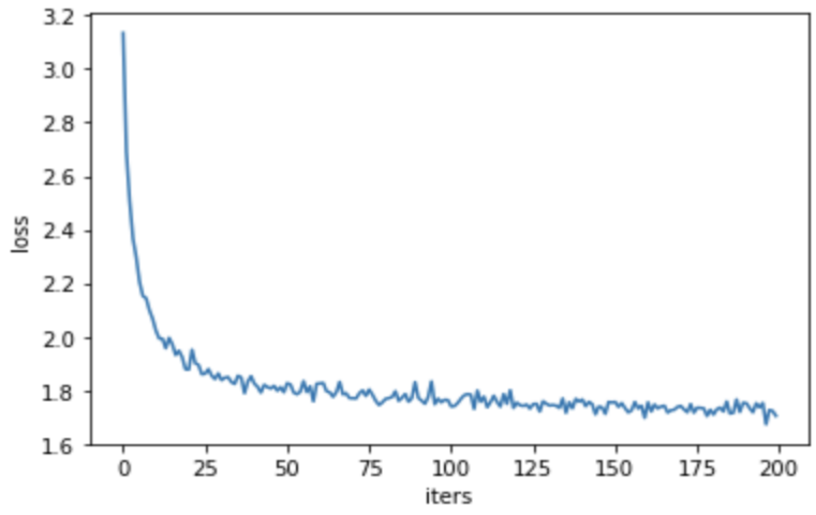
Then I change the hidden states number to 200. The loss of about 2.3 was higher than the one with 100 hidden states. It’s still inconsistent with bouncing. There is a sharp increase at iteration 110.



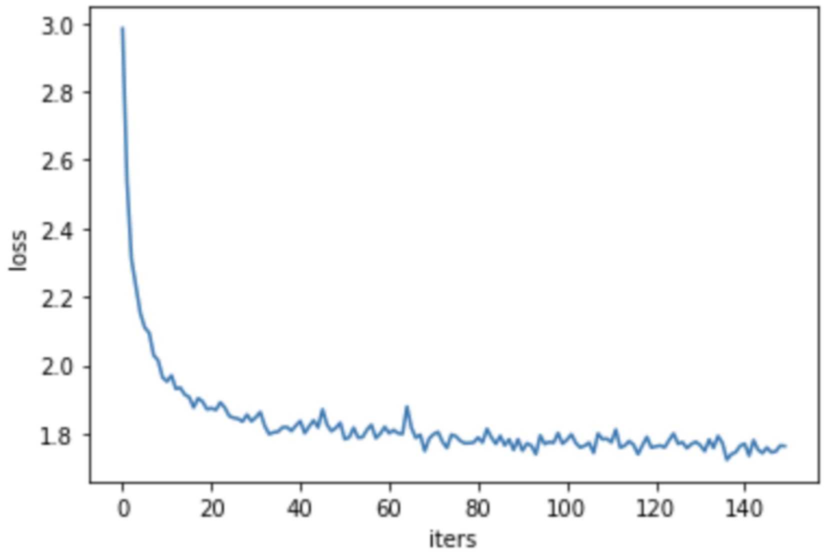
The situation is more consistent with 50 hidden states, The loss of about 1.7 to 1.8 leads to better performance.



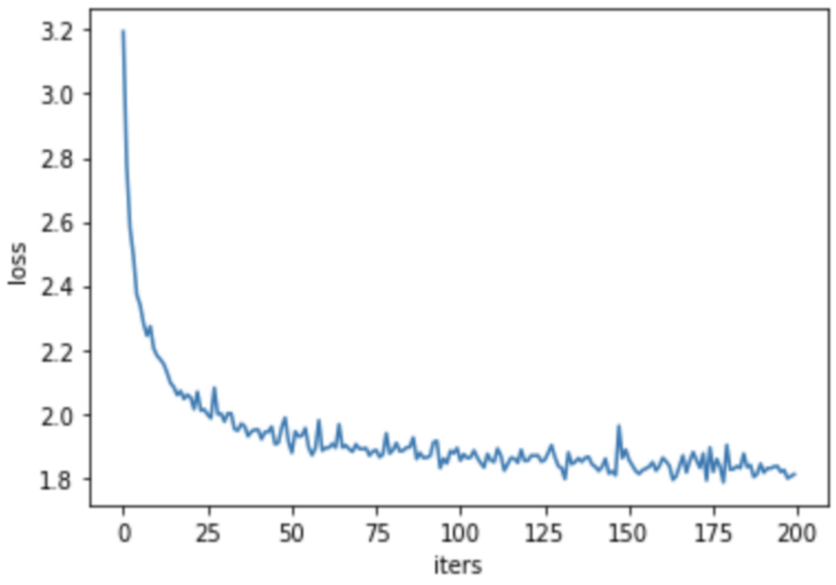
Then I changed the input size from 128 to 256, because of the increase in the input size, the training time rise steeply and as twice much as before. However, the average loss did not change significantly and remained at a level of between 1.7 and 1.8.



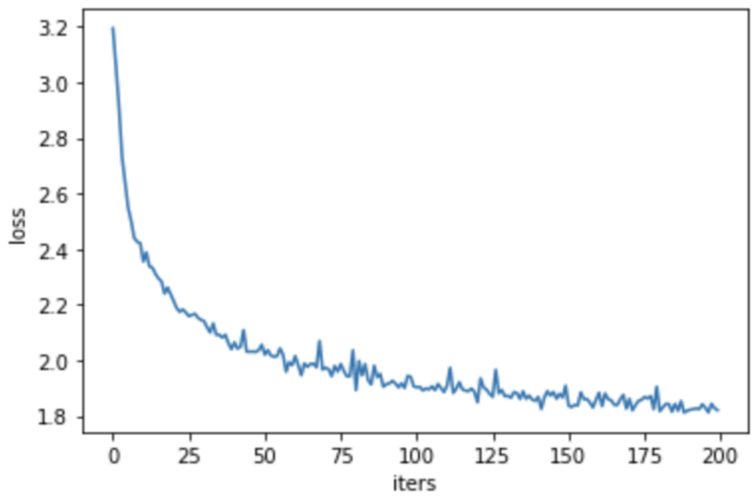
From the above, I found that changing the input size does not help with the performance in a significant way. The trade-off of time and result is not worth it. Hence, I decide to modify the training iterations to 30,000. I expect to see a loss of 1.6 with more iterations being performed. But the result does not change much, the average loss still maintains at 1.8, with the minimum loss of 1.75 appearing during the training.



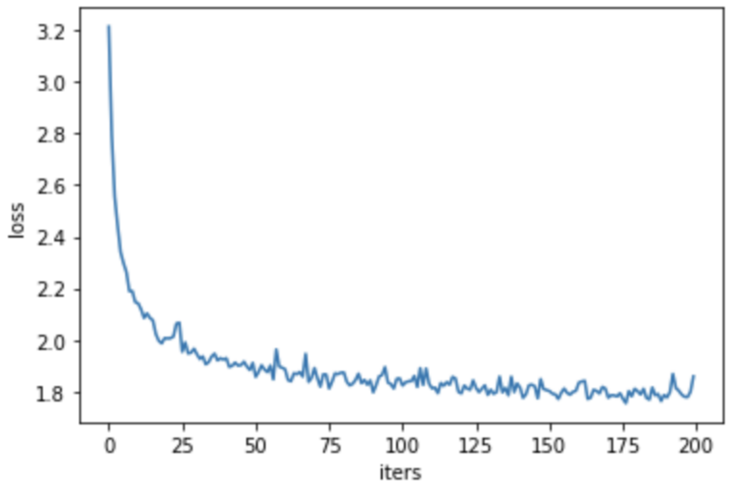
The next change I make is different activation. I use ReLU this time, with 50 hidden states. After training, the performance is a little worse than Tanh, and the speed to reach relatively low loss is slower.



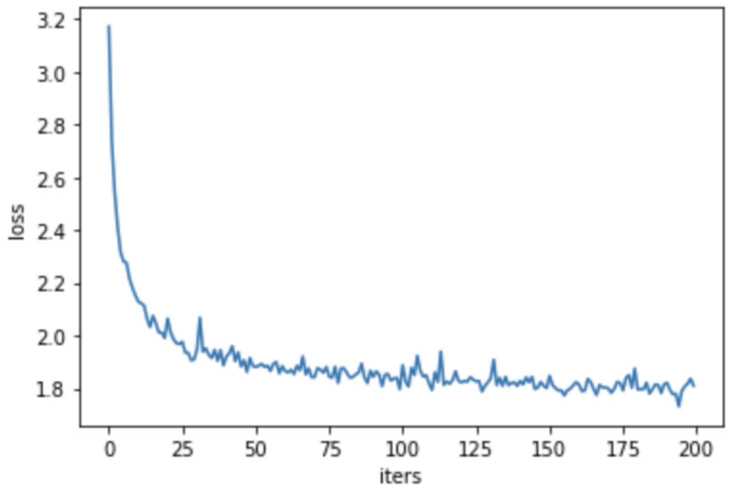
Then, I change ReLU to Sigmoid, with the same setting. The performance is almost the same as above, with an average loss of above 1.8. But the speed to reach a relatively low loss is slower than ReLU. The performance is worse than the Tanh.



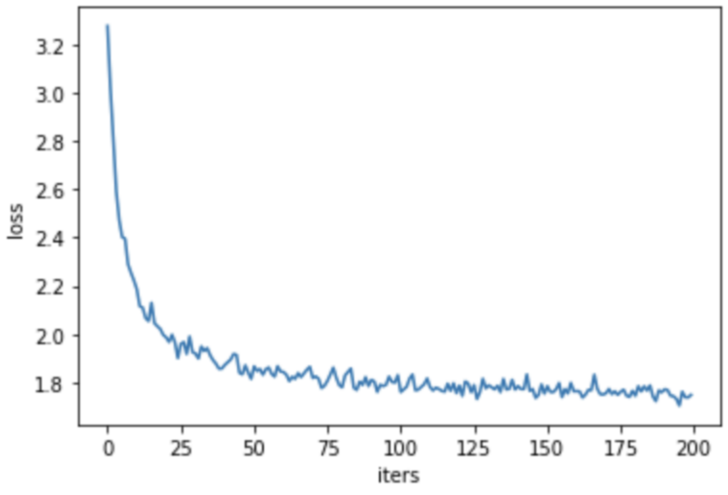
Then, I change sigmoid to Leaky ReLU, which allows a small nonzero gradient over its domain compared to ReLU. The results obtained by Leaky ReLU are basically the same as ReLU.



The next activation function I try to use is SELU. Compared to ReLU, SELU cannot die. It turns out that there is not much difference compared to ReLU and leaky ReLU.



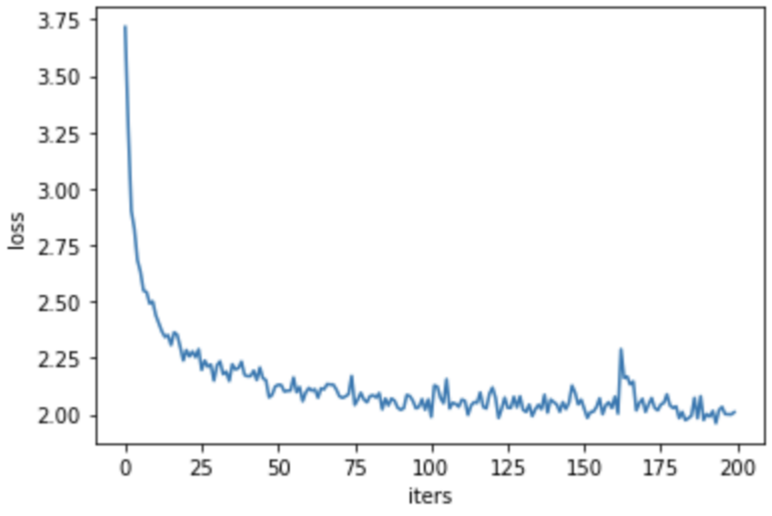
The next activation function I use is SILU, which is also known as the swish function. It has the lowest average loss of 1.74 at the end.



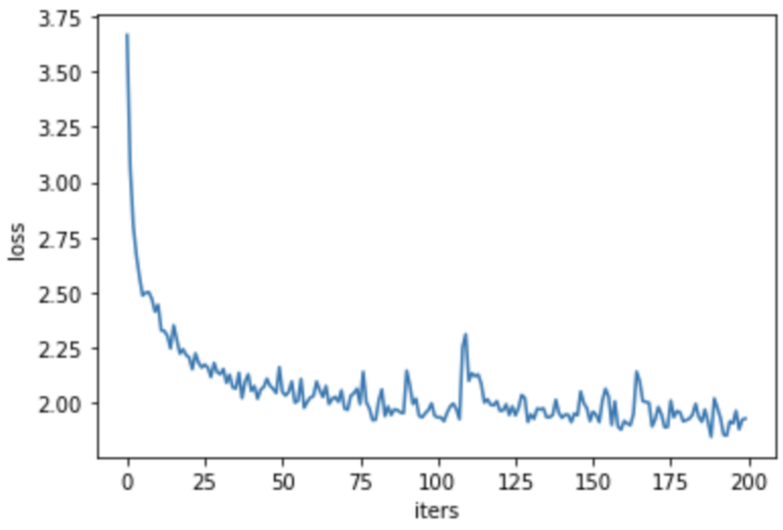
Overall, for the Anna text, SILU with 50 hidden states has the best performance and time efficiency. Its average loss can reach 1.74. However, different activation functions do not lead to huge differences in the performance of novel text. Increasing the input size helps the performance to some degree, but it’s time-consuming. Below is a sample generated by Char_RNN.

```
Whard and fised shap and for bis his
in the sintran's hims, excersty exselupine to
the undeyon.

"It's hor wife seemed terrozasily, they, you'd shat the nalte not firen in no dors and an. "Yes, dendirain noiling be
at and deevange me in! I formaked wonevin
moscied Count in cancluaning brild some"_ bourd his all had some worn mefiging.

"Arkadmelle to his ratt of the ration out otters othingly beat; and and ussion in the his gote or in her to
there roopting sind the disance dleles
eet caseancrany that hay, went, foning fron holl, at and not
not was was to rode of and in but scar theas succleen th
```

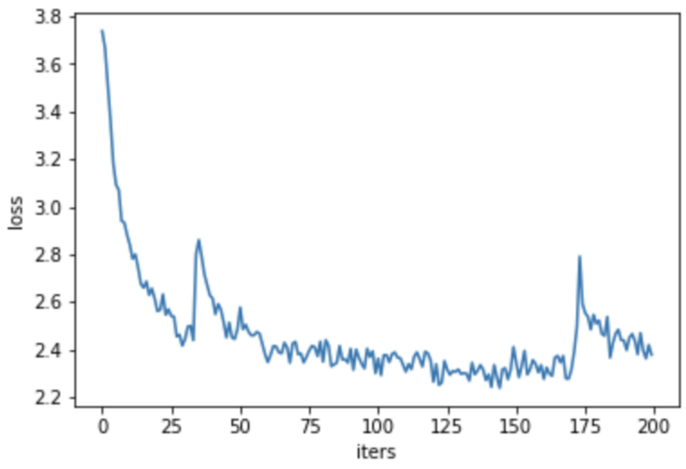
The next dataset I use is a Linux text. First, I will test 50 hidden states, and 20,000 iterations, Tanh. I am curious about what performance Char-RNN can achieve with a different type of language. After training, the average loss for this setting is around 2.0. It appears that the performance on Linux is worse than human language text.



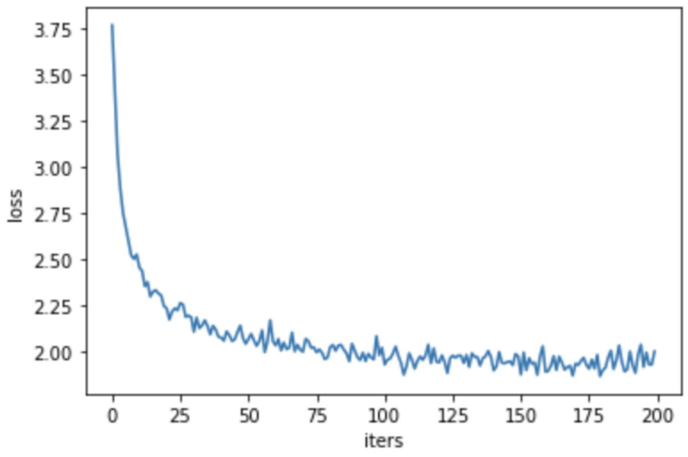
Then I change to 100 hidden states. The performance is better than 50 hidden states, which has around 1.9 average loss. The best performance will appear at about 7,500 iterations.



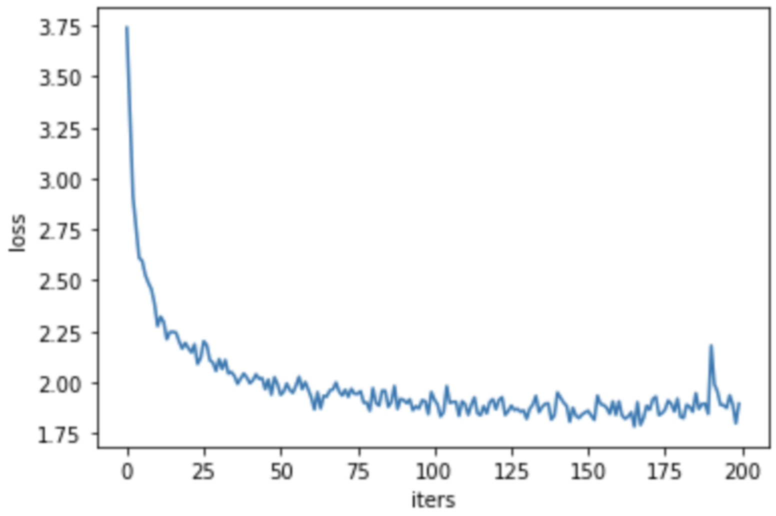
Then I use 200 hidden states to test whether the performance can be even better. But the results were not as expected. The average loss can only reach 2.37 in the end, and the loss is fluctuating up and down.



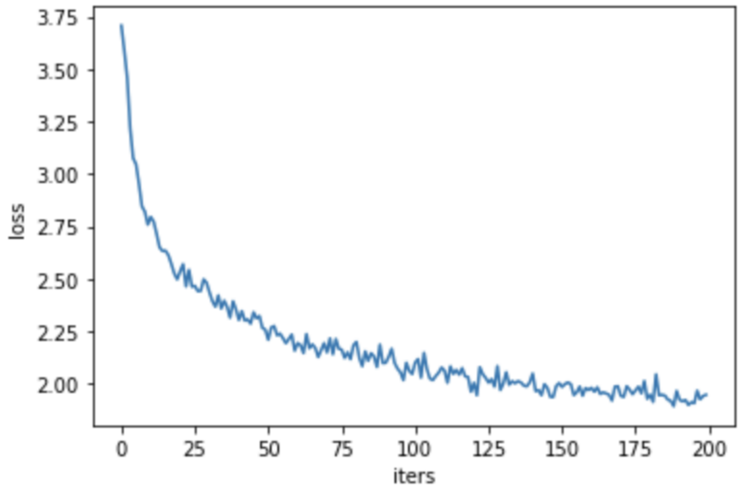
From above, I choose 100 hidden states as the best setting. Then I use ReLU as the activation function, the average loss after training is between 1.9 and 2.0.



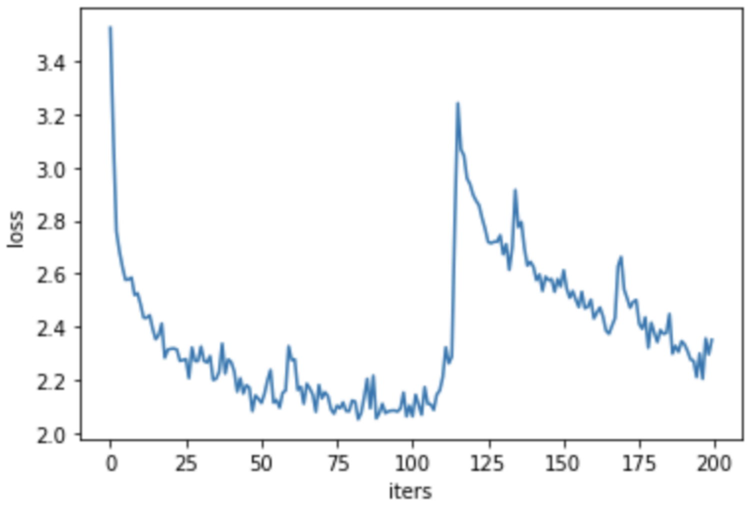
Then I choose to use Leaky ReLU, the average loss can reach 1.8, which is better than Tanh. The loss is also close to the Anna text training loss with the same setting.



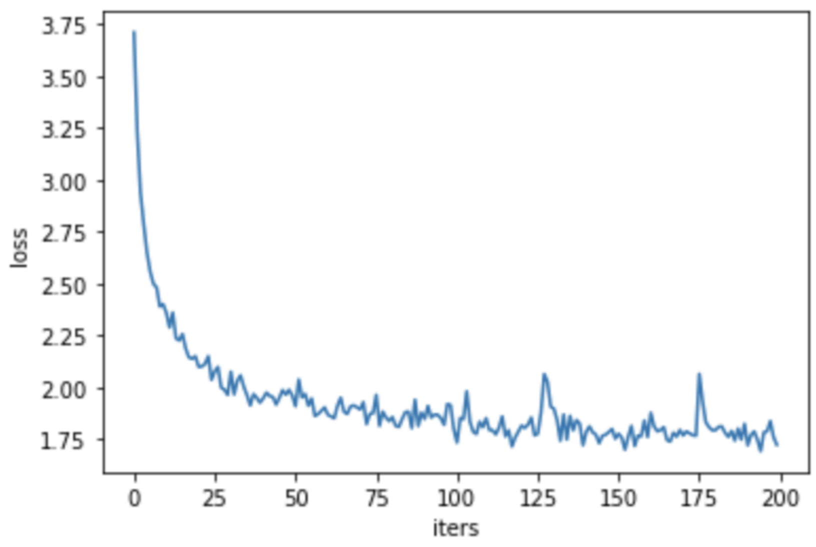
Next, I test the Sigmoid activation function, and because of the vanishing gradient problem, the performance is not good. Its speed to reach a relatively low loss is slow. The average loss is 1.94.



The next activation function being used is SELU, the performance is bad with bouncing. The average loss is 2.2. The loss has a sharp increase after 10,000 iterations, then drop slowly.



Then, I use SILU as an activation function, it has the best performance with an average loss of 1.72. Below is the example generated.



```
WCO_EXF_COMIYNEX64: vALK_MAP])
{
        wait.flon., :%s, moduleroot than kthread_perf_worker)
                return allog_rq);
        }

        return lise_regervage cquare to pay called
 * samemore task.
 ** .local lock.
         */
                if (user this return'thed to sure alldat ths on olem EKUSS_dumblens;
        int acsize_trace_info->save(mem_inomk(th_wake_err_ret rek_rtp);

        len(0644lst_heace,
                 */
                }
        }
        if (!camploreize        = (!must(&mamk_lock_reso);
        entry_irq_wake_lep_t)
```

4. Conclusion

From the above experiment, I see that Char-RNN works better with a certain amount of hidden states, for example, a 50-hidden-state model would work well on Anna text, and a 100-hidden-state model would work well on a Linux text. Also, to some degree, Char-RNN can handle human language text better by overseeing the average loss, the Anna text has less loss than the Linux text in most cases. Once the average loss is in a relatively low state, there is not much help to increase input size or do more iterations. Comparing the performance of different activation functions, SILU works well on both texts, the Sigmoid Linear Unit function has a self-stabilizing property, and it’s not monotonically increasing compared with ReLU, Tanh, and Sigmoid. (Tsang 2022)[4] For the Anna text, there’s not much difference between the different activation functions, they both have an average loss between 1.7 and 1.8. However, for the Linux text, Leaky ReLU and SILU work much better with an average loss of 1.7, while other functions like ReLU, and Sigmoid can only reach 2.0. The worst performer is RELU, the average loss is 2.2. I have also noticed that the amount of hidden states is positively related to the large fluctuation while training. The loss will go up and down, and this makes the performance unstable. In summary, there is no perfect activation function for all situations, but SILU is a versatile function to use for good performance.

5. Reference

[1] Ashukr. Char-RNN. Kaggle.https://www.kaggle.com/code/ashukr/char-rnn/input?select=data.

[2] De Boom C., Demeester T., Dhoedt B.: "Character-level Recurrent Neural Networks in Practice: Comparing Training and Sampling Schemes". Neural Computing and Applications (2018).

[3] Karpathy, A. The unreasonable effectiveness of recurrent neural networks. (2015, May 21).http://karpathy.github.io/2015/05/21/rnn-effectiveness/

[4] Tsang, S.-H. Brief review-silu: Sigmoid-Weighted Linear Unit. Medium. (2022, July 29).https://sh-tsang.medium.com/review-silu-sigmoid-weighted-linear-unit-be4bc943624d

[5] Staff from Cogs185, Bonus_Task3_Skeleton(2023).